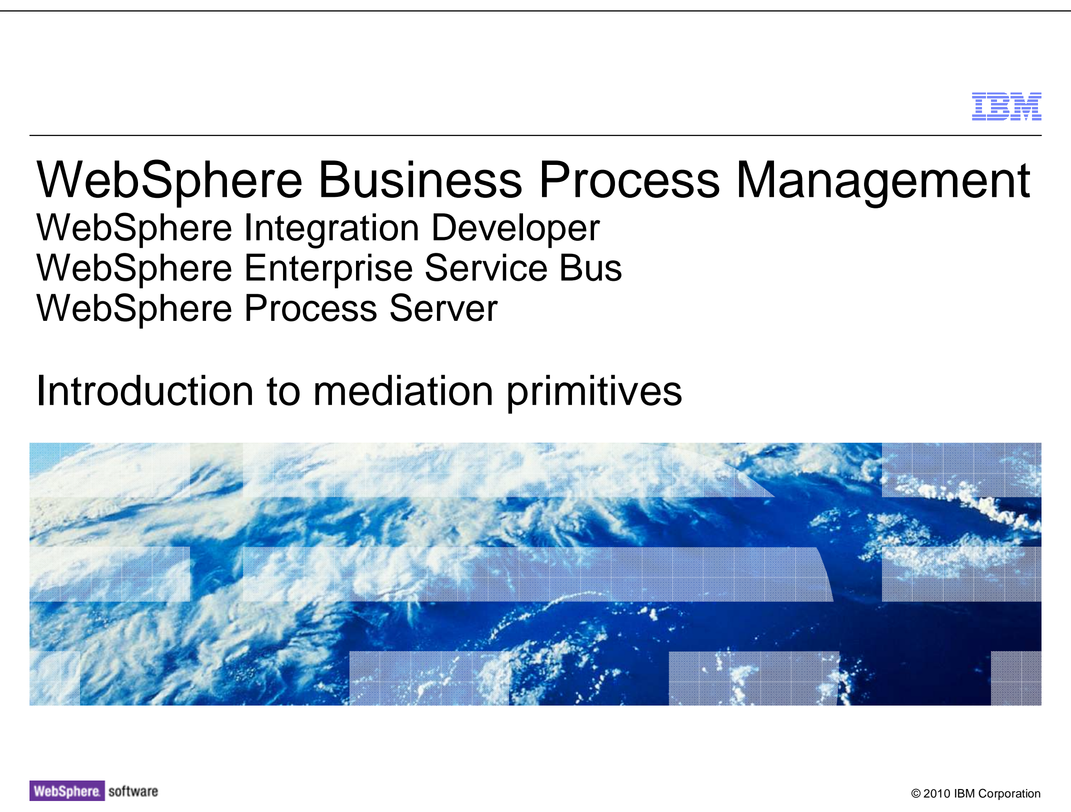
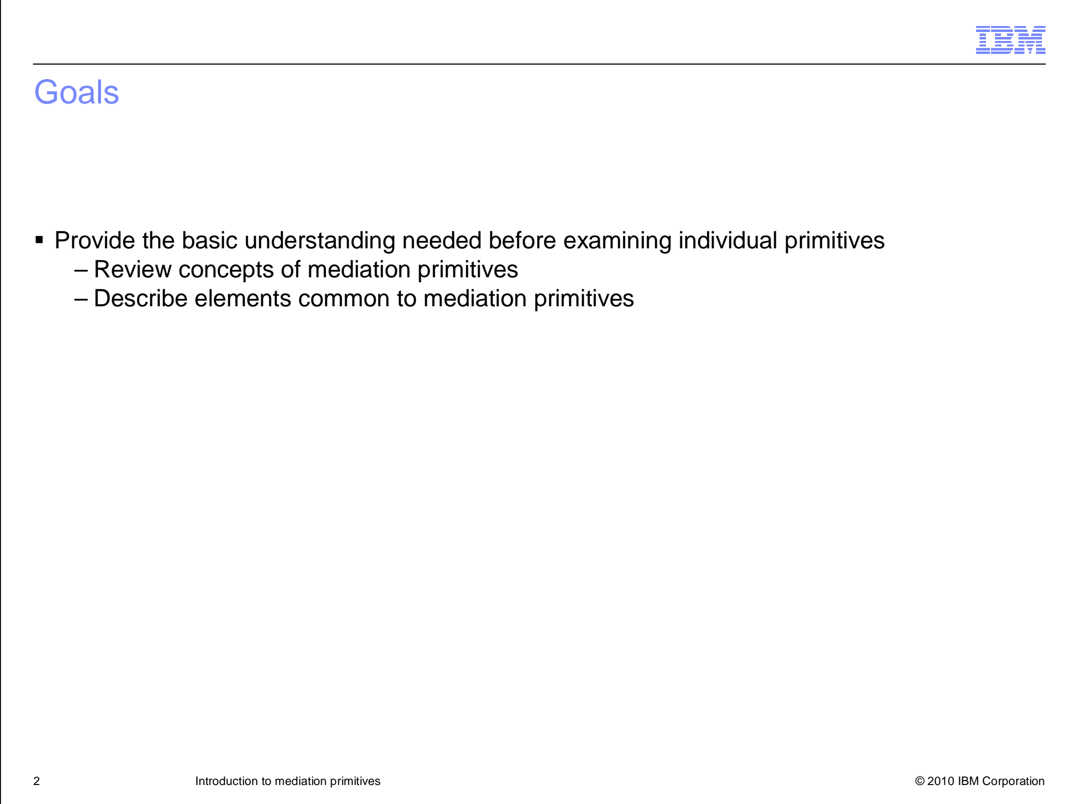


This presentation introduces mediation primitives and discusses those aspects that are common to mediation primitives in general.



The goal of this presentation is to provide a basic understanding before discussing each of the individual mediation primitives. This is done by reviewing the concepts of mediation primitives and describing the elements that are common across many or all of the primitives.

IBM

Place of mediation primitives in the big picture

- Mediation modules:
  - Mediate messages flowing between service requestors and providers
    - Handle protocol transformations
    - Update content of the message
    - Modify format of the message
    - Dynamically route service requests and responses
  - Contain a mediation flow component
- Mediation flow components:
  - Used to define the mediation flow logic
  - Unique flow logic defined for every operation of a service interface
- Mediation primitives
  - Wiring of primitives is used to construct the logic of a mediation flow
  - Each primitive performs some specific part of the flow logic
    - Each primitive type performs some predefined function
    - The predefined function is configured for each instance through the use of properties

 © 2010 IBM Corporation

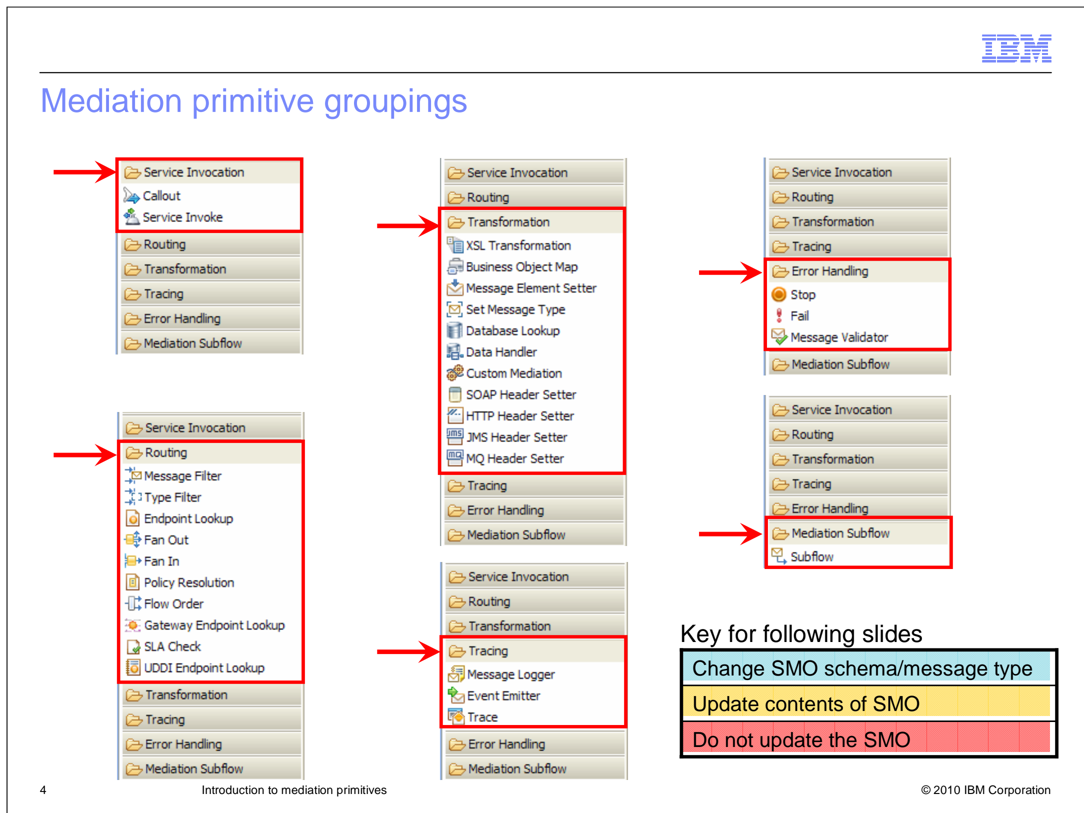
In order to understand mediation primitives, it is important to understand where they fit into the big picture of mediations.

Starting at the highest level of abstraction, there are mediation modules whose function is to mediate messages flowing between service requestors and service providers. Mediating a message might involve handling protocol transformations, updating the content of the message, modifying the format of the message or dynamically routing the message to an appropriate service provider.

The mediation module contains a mediation flow component, which is where the overall logic for the mediation is defined. For every operation defined on an input interface there is unique mediation flow logic for the operation's request and response. The flow logic is defined within the mediation flow component using mediation primitives.

Each type of mediation primitive provides some predefined functional capability. Each instance of a mediation primitive has its predefined function configured through the use of properties. The property settings define how the primitive explicitly behaves in this specific instance. The overall logic of the flow is defined by wiring these configured mediation primitives together into a logical flow.

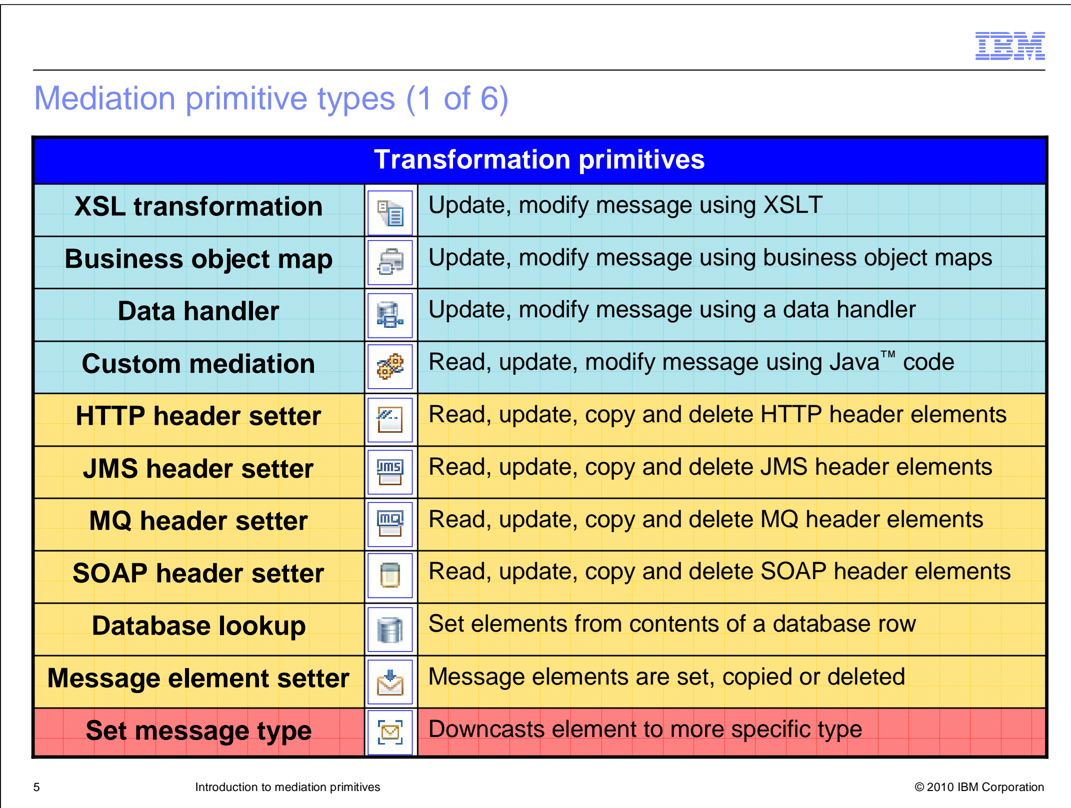
To summarize, the highest level of a mediation is the mediation module, which contains a mediation flow component, which contains mediation flows, which are composed of mediation primitives wired together to define the logic.



The next few slides introduce the various mediation primitive types. There are various ways that mediation primitive types can be organized or classified. In WebSphere® Integration Developer, they are categorized into broad groupings based on the primary function of the primitive.

These groupings are shown using screen captures from the mediation flow editor palette. In the upper left, you see the service invocation primitives which are used to interface with external services. Below them are the routing primitives. They are used to either control the path within a flow or identify endpoints for external services. In the center are the transformation primitives which are used to access and modify data in the SMO. Some can also manipulate the schema of the SMO, thus changing the message type. Below them are the tracing primitives, which write data extracted from the SMO to an external source such as a log or event queue. In the upper right are the error handling primitives that are used to check for errors or control the flow in error situations. Finally, below them is the mediation subflow. A subflow can be thought of as a user defined primitive. They are used to encapsulate the logic of a flow, enabling it to be reused in other flows as if it were a primitive.

In the slides that follow, the organization just described is used as the highest level of organization. In addition, another way to group primitives is according to their behavior and abilities for updating the SMO as it flows through the mediation. The following slides address that as well, using color coding as an indicator, which you can see in the key on the bottom right. Primitives which can change message type are shown in blue, those that update the SMO without changing the message type are shown in yellow and finally those that do not update the SMO are shown in red.



This slide looks at transformation primitives, starting with those that can change the message type, which are shown in blue. This is the truest sense of transformation, as the message type of the SMO is being transformed.

The **XSL transformation** primitive is used to update or transform messages using XSLT, which can be used to change the format of the message. An example of when the format needs to change is when the target provider has a different interface than the incoming message.

The **business object map** primitive is very similar in function to the XSL transformation primitive, but it uses business object maps rather than XSLT to perform the transformation. These are the same business object maps that are used in WebSphere Process Server to perform parameter mapping within an interface map. As a result, change logging and the relationship service are enabled.

The **data handler** primitive allows you to configure a data handler to transform the message. A data handler can be one supplied with the product or one that you provide. The data handler and any associated configuration data are supplied as a resource configuration that defines the exact behavior of the transformation to be performed. These data handlers and resource configurations are the same as those used with SCA imports and exports.

The **custom mediation** primitive is used to do any message processing not covered by the other mediation primitives. This is done through Java code that can be written as a visual snippet or as a Java snippet.

IBM

Mediation primitive types (2 of 6)

Transformation primitives		
XSL transformation		Update, modify message using XSLT
Business object map		Update, modify message using business object maps
Data handler		Update, modify message using a data handler
Custom mediation		Read, update, modify message using Java code
HTTP header setter		Read, update, copy and delete HTTP header elements
JMS header setter		Read, update, copy and delete JMS header elements
MQ header setter		Read, update, copy and delete MQ header elements
SOAP header setter		Read, update, copy and delete SOAP header elements
Database lookup		Set elements from contents of a database row
Message element setter		Message elements are set, copied or deleted
Set message type		Downcasts element to more specific type

6 Introduction to mediation primitives © 2010 IBM Corporation

Continuing with the transformation primitives, the next grouping are those that update the contents of the SMO but do not actually transform the message type.

The first four, the header setter primitives, are all similar in nature. They provide you with an easy way to access protocol specific headers within the header section of the SMO. Although these headers can be accessed with other mediation primitives, the header setter primitives make it much easier because they are aware of how these protocol specific headers are structured. This allows you to focus on those aspects of the header you are interested in reading or updating, without having to be concerned with specifically how they are represented in the SMO.

The **HTTP header setter** primitive enables you to access HTTP headers, which are grouped into control elements, standard elements and user elements.

The **JMS header setter** primitive enables you to access JMS headers, which are grouped as standard elements and user elements.

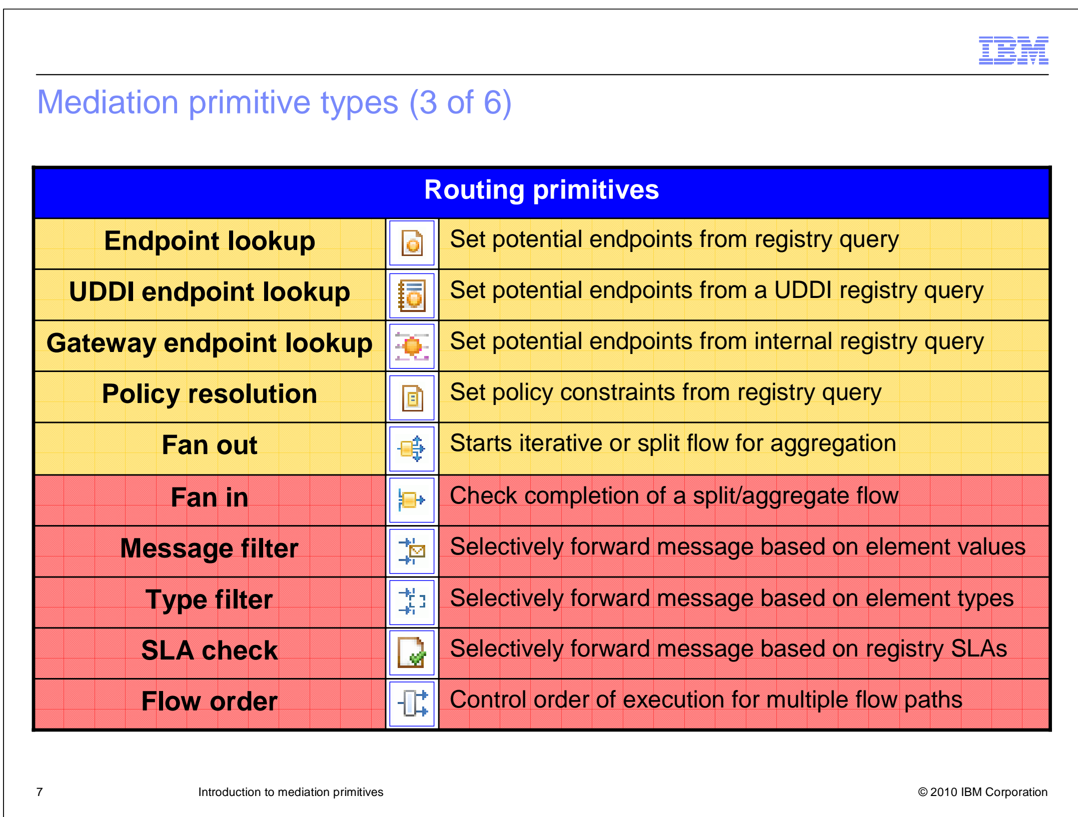
The **MQ header setter** primitive enables you to access MQ headers. The headers exposed are the message descriptor, the CICS® bridge header, the IMS™ information header and the rules and formatting header two.

The **SOAP header setter** primitive enables you to access SOAP headers. Any XSD containing an element can be a SOAP header. Some are provided by standards such as WS-Security and others are user defined.

The **database lookup** primitive is used to access information from a database and insert it into the message. A field in the message is used as a key for the database access and selected fields from the resulting database row can be placed into the message.

The **message element setter** primitive can be configured to update elements of the SMO. Individual elements can be set to a specific value or can have their value deleted. Individual elements or sub-trees in the SMO can be set by copying the values from another location in the SMO. Arrays in the SMO can have an element appended.

The **set message type** primitive does not update the SMO contents, but is used to augment the message type. It is used in conjunction with loosely typed elements in the SMO, such as an XSD:anyType, allowing it to be downcast to a more specific type. This enables tools, such as the simple XPath expression builder, to represent the element to you as the more specific type.



This next grouping of primitives are those that are categorized as routing primitives. The first group are those that update the SMO.

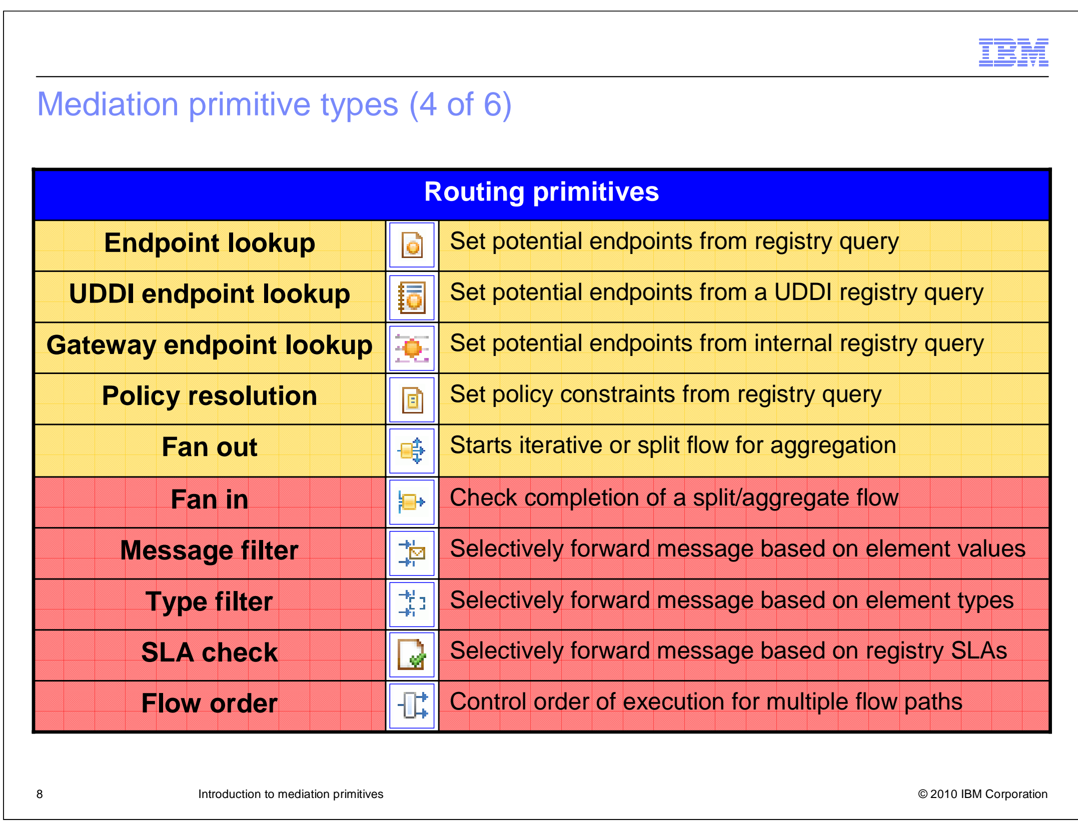
The **endpoint lookup** primitive is used to perform a query of the WebSphere Service Registry and Repository. The SMOHeader section of the SMO is updated with potential service endpoints that can be used by a callout node or service invoke primitive. Looked at from this perspective, the endpoint lookup is not actually doing the routing, but rather is providing the information used later in the flow to perform routing.

The **UDDI endpoint lookup** primitive performs the same function as the endpoint lookup, with the exception that the registry query is to a UDDI registry rather than WebSphere Service Registry and Repository

The **gateway endpoint lookup** primitive is used in proxy gateway scenarios. It performs a query to an internal registry to access potential service endpoints based on a virtual service name obtained from the message. It updates the SMO header with the endpoint information, similar to the operation of the other two endpoint lookup primitives.

The **policy resolution** primitive performs a query of the WebSphere Service Registry and Repository. Based on a conditional query, it looks up policy information and updates the dynamicProperty section of the SMO with property values. These property values are then used by subsequent primitives in the flow to influence their configured behavior.

The **fan out** primitive is used to start an iteration for a message splitting and aggregation flow. It can be used to process an array of repeating elements within the SMO and can also be used as the head of a flow with multiple paths. For a simple splitting scenario the fan out can be used without a fan in, but for a splitting and aggregation scenario it has an associated fan in. When used with an aggregation scenario, the primitive updates the FanOutContext section of the SMO.



Routing primitives		
Endpoint lookup		Set potential endpoints from registry query
UDDI endpoint lookup		Set potential endpoints from a UDDI registry query
Gateway endpoint lookup		Set potential endpoints from internal registry query
Policy resolution		Set policy constraints from registry query
Fan out		Starts iterative or split flow for aggregation
Fan in		Check completion of a split/aggregate flow
Message filter		Selectively forward message based on element values
Type filter		Selectively forward message based on element types
SLA check		Selectively forward message based on registry SLAs
Flow order		Control order of execution for multiple flow paths

Continuing with the routing primitives, the next group are those that do not update the SMO.

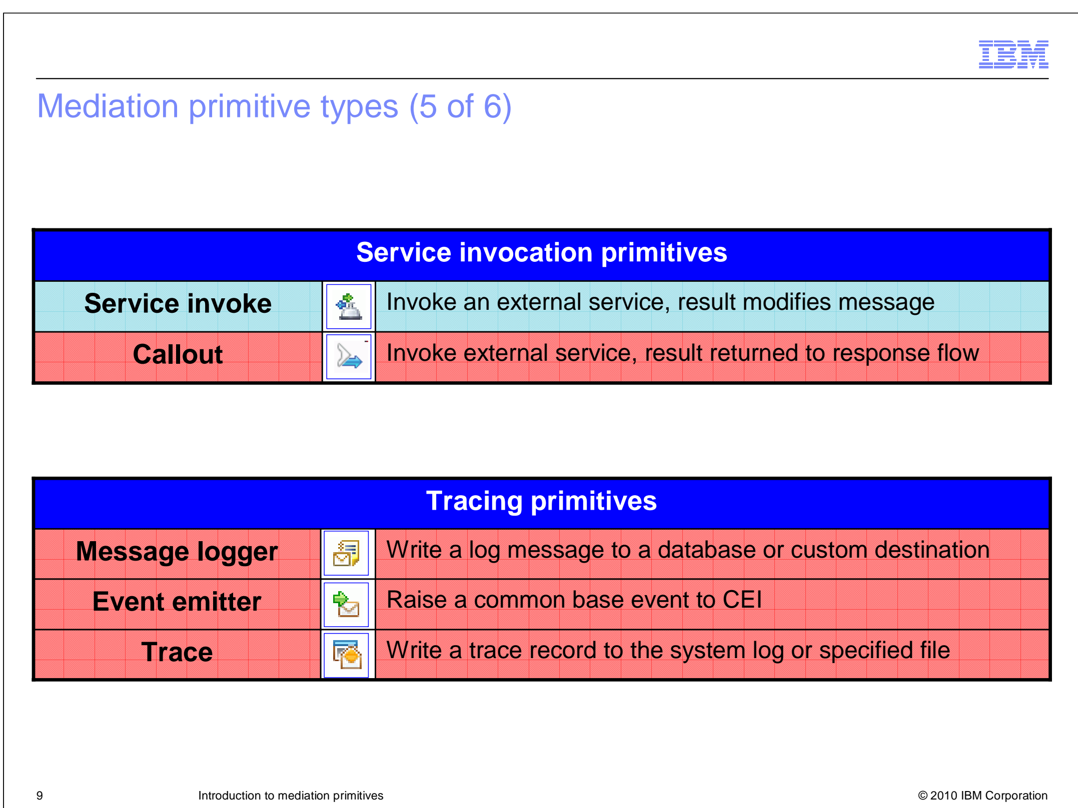
The **fan in** primitive is always used in conjunction with a fan out primitive as part of a message splitting and aggregation scenario. The primitive controls the flow based on the state of the flow and the fan in configuration information. Either the flow returns to the fan out for another iteration or the flow proceeds from the fan in, passing the aggregated results.

The **message filter** primitive is used to modify the path through a flow by selectively forwarding the message based on values of elements within the SMO. The primitive contains a table of simple XPath expressions, each associated with an output terminal defining where the message is forwarded. The message is forwarded based on the results from evaluation of the XPath expressions.

The **type filter** primitive is very similar to the message filter. It selectively forwards the message based on the evaluation of element types within the SMO. The primitive contains a table with elements and associated types, each associated with an output terminal defining where the message is forwarded. The message is forwarded when an element in the SMO is of the same type as that defined in the table.

The **SLA check** primitive provides a mechanism to incorporate service level agreements into mediation flows. The primitive makes a call to the WebSphere Service Registry and Repository which contains configuration information for service level agreements. Different paths are then taken through the flow based on the response from the registry. The response indicates whether the request conforms to the configured service level agreements.

The **flow order** primitive is used in the case where you normally have multiple paths wired to an out terminal. By inserting the flow order primitive, you can explicitly define the order in which those paths are executed.



This next grouping are the service invocation primitives.

The **service invoke** primitive is used to make a call from within a mediation flow to an external service defined on the mediation module assembly. The service can be defined by a Java component or by an import. This primitive changes the message type of the service message object. The input terminal message type conforms to the input to the service and the output terminal message type conforms to the output from the service.

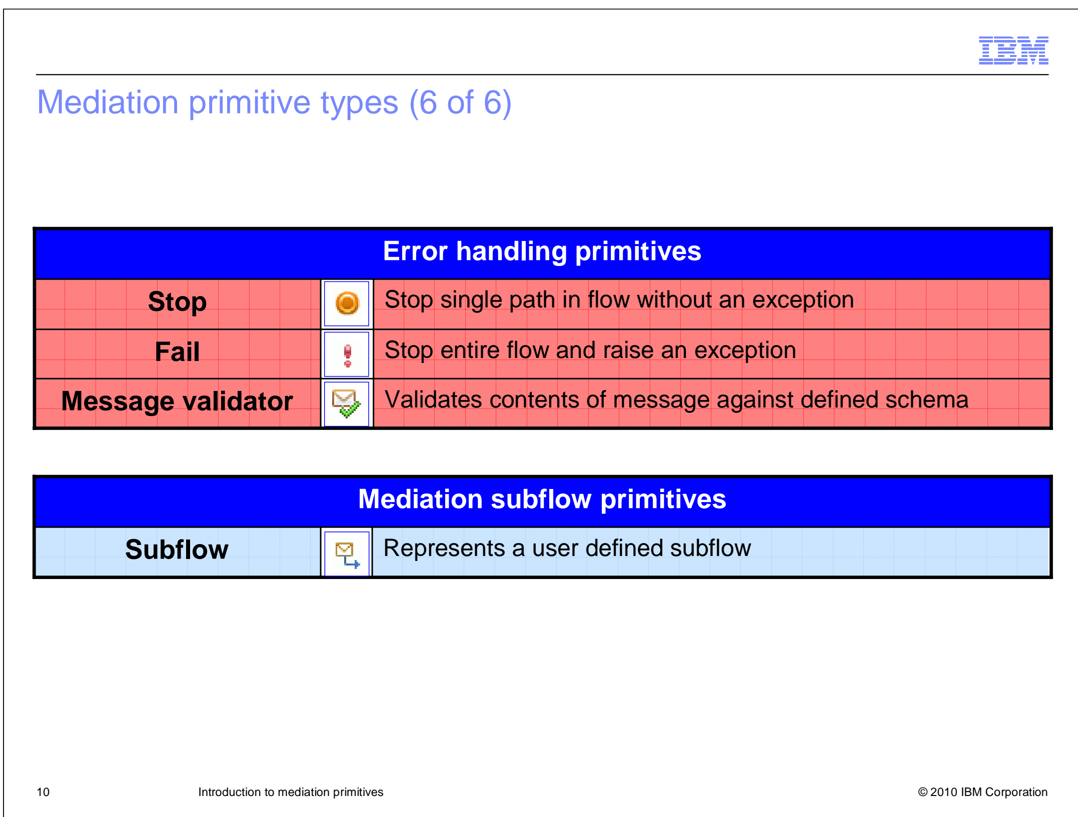
The **callout** is actually not a primitive, rather it is a node in a request flow. It is the only non-primitive that is made available on the mediation flow editor palette. It is handled similar to a primitive in that it is dropped onto the canvas and wired into the flow. It has an input terminal, but no output terminal. The purpose of the callout it to make a call to an external service, ending the request flow. The response from the external service initiates the response flow in the mediation. It does not change the service message object as the SMO is not passed beyond the callout. A new copy of the SMO is created when the response flow is initiated.

The next grouping are the tracing primitives. None of these primitives updates the SMO.

The **message logger** that is used to log all or part of the contents of the message. The primitive provides two different options. The first option logs the message to a message log database which is identified through configuration of the primitive. The other option allows you to specify a custom logging implementation based on the Java logging APIs. A default implementation of this is provided that logs to a file.

The **event emitter** primitive is used to raise an event containing all or part of the contents of the message. The event is emitted as a common base event which is handled by the common event infrastructure.

The **trace** primitive writes all or part of the message as a trace record to either the system log or to a specified tracing file.



The next category are the error handling primitives.

The **stop** primitive is used to stop an individual path through the mediation flow without raising an exception or affecting other paths through the flow.

The **fail** primitive is used for error conditions and will stop the entire mediation flow and cause an exception to be raised.

The **message validator** primitive enables you to designate a portion of the SMO to check, ensuring that the actual data is consistent with the schema for that portion of the SMO. Different out terminals are taken depending upon the success or failure of the validation.

Finally, there is the mediation **subflow** primitive which represents flow logic that you have provided as a reusable subflow. A mediation subflow is a preconfigured set of mediation primitives that are wired together to create a common pattern or use case. The logic of the subflow determines if the SMO contents are update and if the message type is changed.

IBM

Mediation primitive input and output terminals

- Terminals:
  - Define a mediation primitive's input and output message type
  - Message types define the content of the service message object body
- Input terminals
  - Defines input message type
  - Generally one per primitive (with a few exceptions)
- Output terminals
  - Defines output message type
  - Zero, one or more output terminals per primitive, either fixed or variable
  - Possibly required to have same message type as input terminal
- Fail terminal
  - Used when a primitive fails during the flow
  - Message type must be the same as input terminal message type
  - Propagates the original message updated to contain failure information

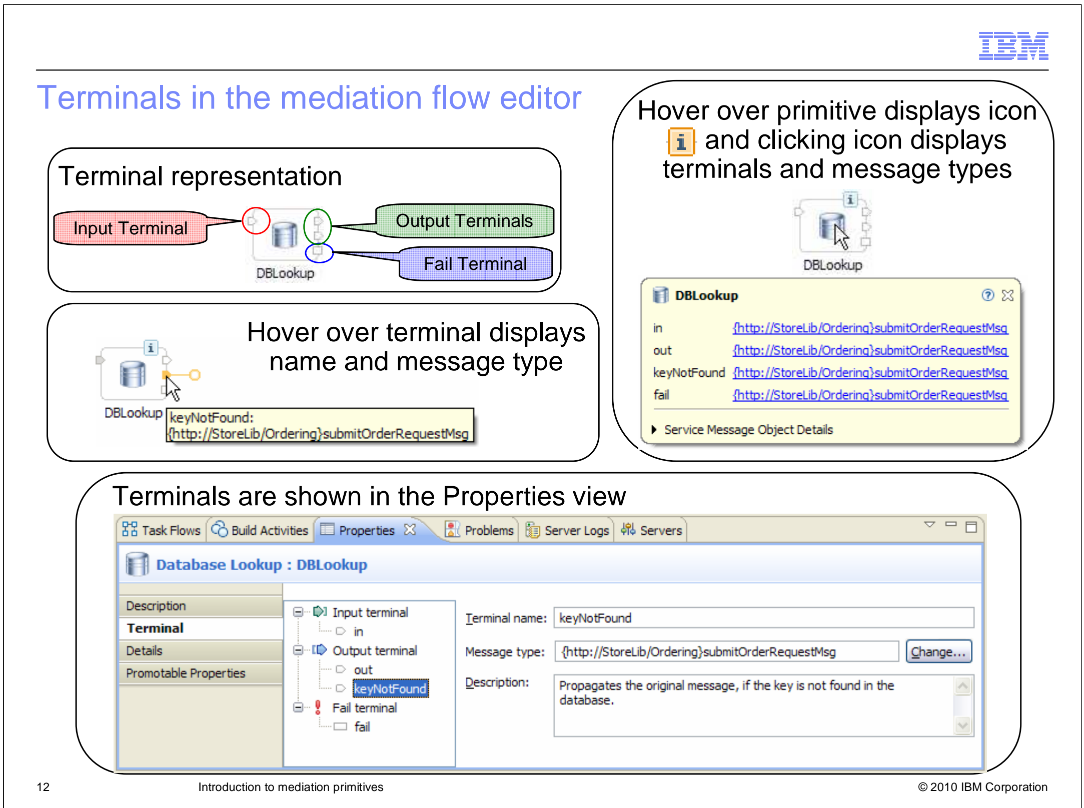
11 Introduction to mediation primitives © 2010 IBM Corporation

All mediation primitives have terminals, which are used to define the input and output of the primitive, identifying the message type that flows through the terminal. The message type is defined by the structure of the service message object body that is present in that part of the flow.

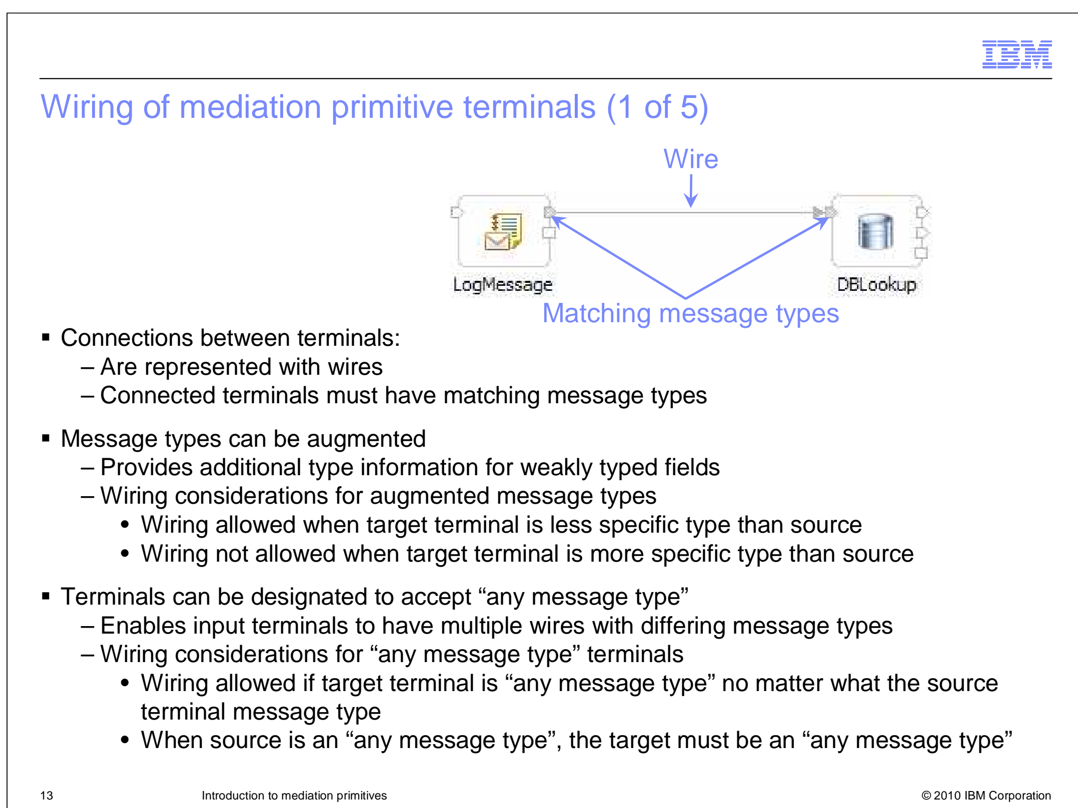
Typically, there is just one input terminal per primitive and it defines the input message type. Exceptions to this are the custom mediation primitive, allowing a variable number of input terminals, and the fan in primitive, which has two defined input terminals.

An output terminal defines the output message type. The number of output terminals varies by the primitive type. A primitive type can have zero, one, two or a variable number of output terminals. For many mediation primitives, the output terminal must be of the same message type as the input terminal. This is because the primitive is not capable of changing the structure of the SMO body. However, for primitives that can change the SMO body structure, the output terminal can be for a different message type.

The fail terminal is used when the mediation primitive fails in some way while processing the message. Because the original message is propagated when there is a failure, the fail terminal is always for the same message type as the input terminal. The message is updated to contain information about the failure.



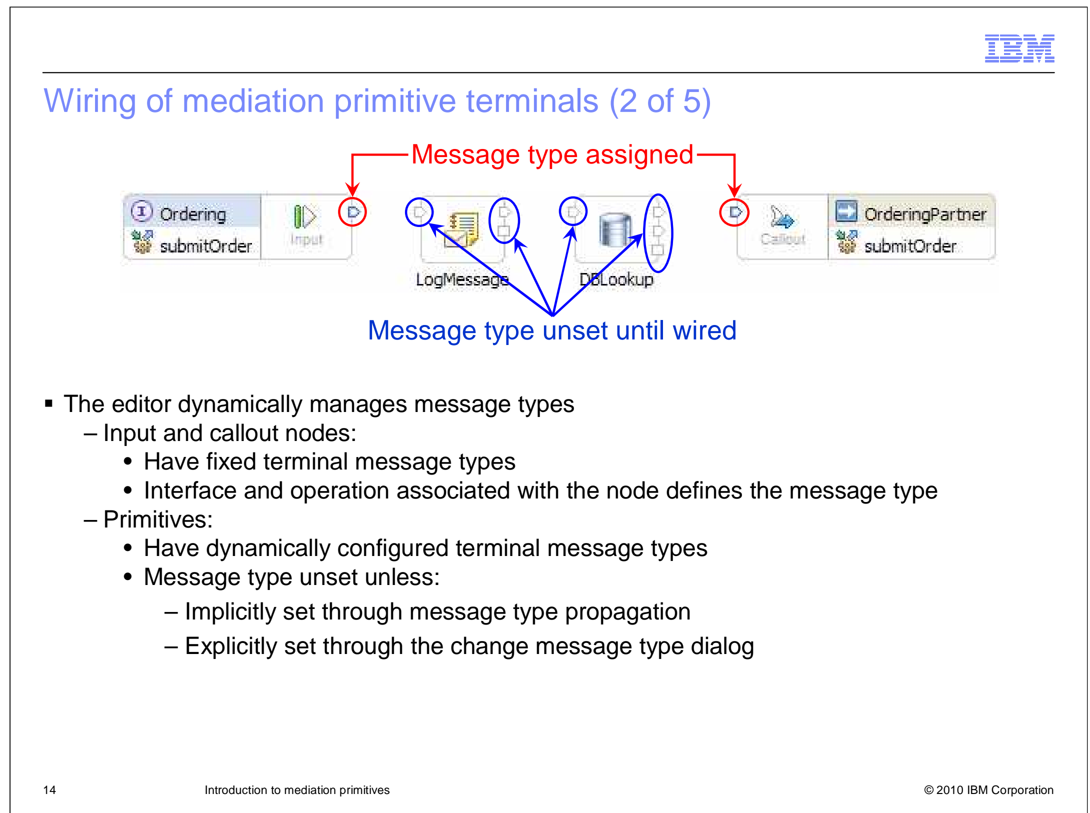
This slide examines how terminals are represented in the mediation flow editor. Starting in the upper left is a mediation primitive. It has an input terminal, which is always on the left side; output terminals, which are on the right side; and the fail terminal, which is the lower terminal on the right side. Notice that the fail terminal has a different shape than either the input or output terminals. Moving down to the illustration in the left center, the behavior when hovering the mouse pointer over a terminal is illustrated. When this is done, a popup opens specifying the name of the terminal and the message type associated with that terminal. The illustration on the upper right shows you what happens when you hover the mouse pointer over the primitive. An icon is shown in the upper right portion of the primitive and clicking on this icon causes a dialog to appear. The dialog provides the name of the primitive, the names of the primitive's terminals and the message type associated with each terminal. This dialog allows you to select the message type for any terminal and expand the SMO details for that message type. From the expanded SMO, you can also launch a dialog to change the message type for that terminal. Finally, on the bottom is a screen capture of the terminal tab in the properties view of the mediation primitive. Selecting any terminal in the list on the left displays the name and message type of the terminal on the right. Notice the **Change…** button next to the message type. This is used to open the change message type dialog for modifying the message type associated with the terminal.



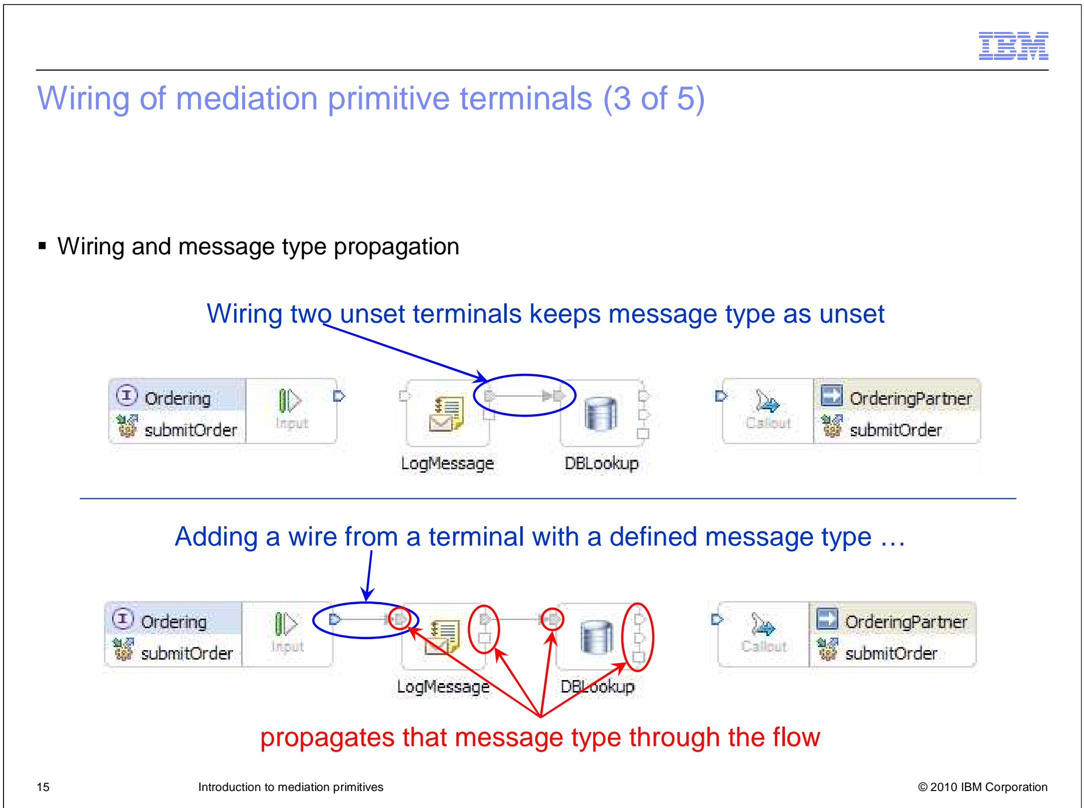
The next few slides look at the behavior of the mediation flow editor relative to the assignment of message types to terminals during the process of wiring a mediation flow. As illustrated in the graphic, connections between terminals are represented with wires. When two terminals are wired together, they must have matching message types.

Message types can be augmented with additional type information. This is used when needing to provide additional type information for weakly typed fields, such as those defined as an xsd:anyType. When wiring terminals that have augmented message types, the two terminals are not required to have exactly the same augmentation. Wiring is allowed when the target terminal has a less specific type than the source terminal. However, if the target has a more specific augmented type than the source, they can not be wired together.

Terminals can also be configured to be "any message type". This allows input terminals to have multiple inbound wires that have differing message types. When wiring to a target terminal set to "any message type", it doesn't matter what the message type of the source terminal is. However, when the source terminal is an "any message type", the target terminal must also be an "any message type".



During the process of wiring terminals together, the editor dynamically manages the message types of the terminals. In the graphic, you see a mediation flow that has an input node on the left, a callout node on the right, and two mediation primitives in the middle. At this point, none of these are wired together. Notice that the nodes have message types assigned to their terminals, whereas the mediation primitives do not yet have message types assigned to their terminals. Terminal message types are either implicitly set through message type propagation or explicitly set using the change message type dialog.



Continuing from the flow shown on the previous slide, this slide examines message type propagation associated with wiring. The top graphic shows a wire added from the output terminal of the LogMessage primitive to the input terminal of the DBLookup primitive. Before wiring, both terminal's have a message type that is unset. After wiring, the message type for these terminals remains as unset.

In the next graphic, the output terminal of the input node is wired to the input terminal of the LogMessage primitive. Because the output terminal of the input node has a specific message type assigned, that message type is dynamically assigned to the input terminal of the LogMessage primitive so that the wire connects terminals of like message type. Since a message logger primitive must have the same output message type as its input message type, the editor dynamically assigns the message type to the output and fail terminals of the LogMessage primitive. Since there is a wire between the output terminal of the LogMessage primitive and the input terminal of the DBLookup primitive, the message type is propagated so that the wire is connecting terminals of like message type. Finally, since a database lookup primitive must have the same output message type as its input message type, the editor dynamically assigns the message type to the output and fail terminals of the DBLookup primitive.

Wiring of mediation primitive terminals (4 of 5)

- Additional message type propagation behaviors
  - Terminals with explicitly assigned message types
    - The change message type dialog can be used to explicitly assign a message type
    - Explicitly assigned message types are not changed by propagation
    - In a situation where propagation normally occurs
      - The message type to be propagated and the explicitly assigned type must be compatible
      - If not compatible:
        - » Wiring might be prevented
        - » Otherwise, incompatible message types warning/error occurs on the wire
  - Removing a wire
    - Terminals whose message type had been dynamically set through propagation are returned to being unset
    - Terminals with explicitly assigned message types are not affected

16 Introduction to mediation primitives © 2010 IBM Corporation

Having covered the basic behaviors of message type propagation, the exceptions to the rule now need to be discussed. The first consideration is for terminals that have an explicitly assigned message type, which has been assigned using the change message type dialog. These terminals are not affected by the message type propagation. However, when in a situation where propagation normally occurs, the explicitly assigned message type and the message type that normally is propagated for this case must be compatible. Generally in the situation where they are not compatible the wiring is prevented. However, in some cases the wiring is allowed or maintained, but an incompatible message types warning or error occurs.

Another consideration is the removal of a wire. When this is done, the propagation is done in reverse. Any terminals whose message type is dependent upon message type propagation through that wire is returned to being unset. This does not have any affect on terminals with explicitly assigned message types.

Wiring of mediation primitive terminals (5 of 5)

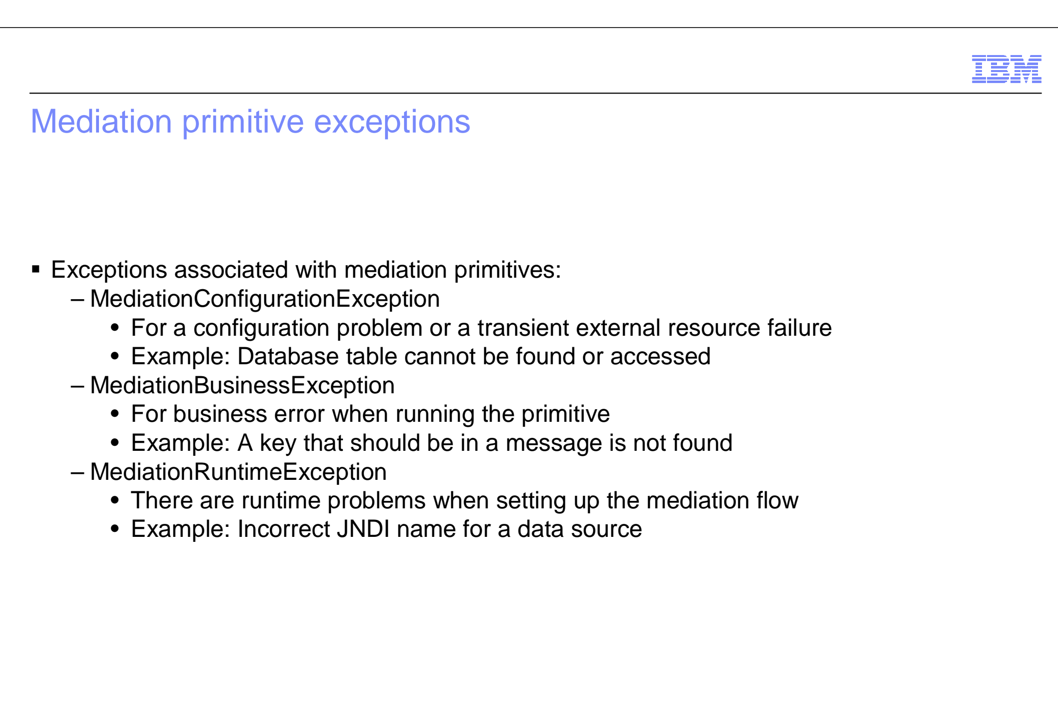
- Primitives with differing message type propagation behavior
  - XSL transformation, business object map, data handler, custom mediation primitives
    - These primitives are capable of changing the message type
    - Message type propagation does not occur between the input and output terminals
    - Message type propagation does occur between the input and fail terminals
  - Service invoke primitives
    - Terminal message types explicitly set based on interface and operation being invoked
    - Not possible to have unset or dynamically set message types
  - Subflows
    - Terminal message types can be explicitly set or left unset by the subflow implementation
    - When unset, propagation between input and output terminals occurs

© 2010 IBM Corporation

This slide looks at some message type propagation behavior differences for specific mediation primitives. There are four transformation primitives that are capable of changing the message type, specifically the XSL transformation, business object map, data handler and custom mediation primitives. No message type propagation occurs between the input and output terminals of these primitives, but message type propagation does occur between the input and fail terminals.

The next primitive to consider is the service invoke. It is not possible to have a service invoke primitive with unset terminal message types. They are always set to the appropriate message types for the interface and operation configured for the service invoke. Therefore, they behave in the same way as terminals with explicitly assigned message types.

Subflows can be implemented to have explicitly defined message types or unset message types for its terminals. When unset, propagation of message type does occur between the input and output terminals.



The specific exceptions that mediation primitives can raise are described here.

A **MediationConfigurationException** is used when there is a configuration problem. It is also used for a transient problem with an external resource, such as not being able to find or access a database.

A **MediationBusinessException** is used when an error that seems to be a business logic problem occurs while running a primitive. An example of this kind of problem is when the database key value configured for a primitive cannot be found in the message.

A **MediationRuntimeException** occurs when there is some kind of problem initializing a mediation flow. An example of this is when the JNDI name for a data source is incorrect.

These exceptions and associated error processing are described in more detail in the next few slides.

IBM

Mediation primitive exception handling (1 of 2)

- Mediation primitive exceptions can be raised:
  - While setting up and initializing the flow
    - MediationRuntimeException
    - MediationConfigurationException
  - While processing the primitive itself
    - MediationConfigurationException
    - MediationBusinessException
    - MediationRuntimeException

 © 2010 IBM Corporation

In order to provide an understanding of the exception processing behavior, it is important to know the different points at which a mediation exception can be raised.

First of all, some initialization is done by the runtime to set up a mediation flow. This initialization occurs before control is given to any mediation primitives. Exceptions that might be raised at this time are the MediationRuntimeException or the MediationConfigurationException, with the MediationRuntimeException being the most common.

Secondly, after initialization of the flow, an exception can be raised during the processing of a mediation primitive. Normally these are a MediationConfigurationException or a MediationBusinessException, but in some cases a MediationRuntimeException can also be raised.

![IBM logo]

Mediation primitive exception handling (2 of 2)

The behavior for processing exceptions is different based on a couple of factors. The first factor is whether the exception is raised during initialization processing or during the processing of a primitive. The second factor is whether the fail terminal for the primitive is wired.

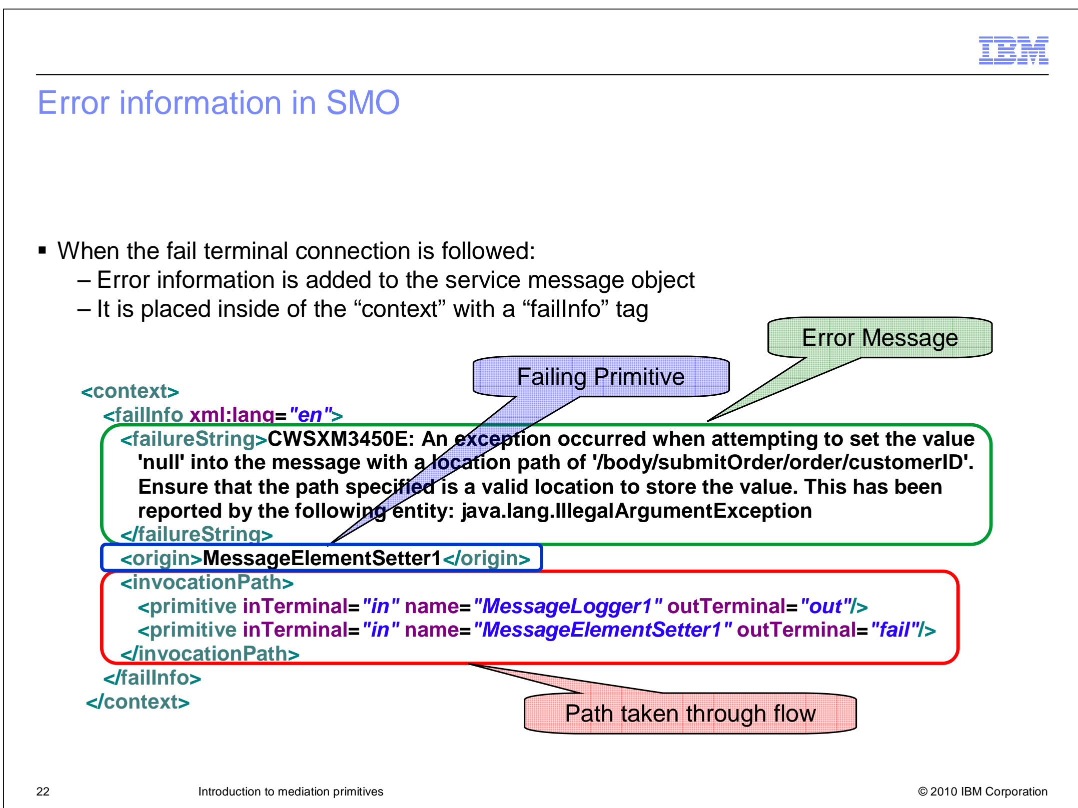
Looking at the slide you can see pseudocode describing the actual behavior. If the exception is raised during the processing of the primitive and the fail terminal is wired to some other primitive or node, then the exception is not logged. The mediation flow continues, following the wire from the fail terminal. In all other cases, the exception causes a log message to be written and the mediation flow to terminate.

Mediation primitive configuration exceptions

- Configuration parameters are checked at different times
  - During setup and initialization of the flow
  - During the processing of the primitive itself
- Handling of these exceptions is therefore different
  - Database Lookup mediation primitive
    - Fail terminal is wired
    - Configuration parameters include:
      - Data source JNDI name
      - Database table name
  - Data source JNDI name not found is discovered during setup, therefore
    - MediationRuntimeException raised
    - Log written
    - Flow terminates
  - Database table not found is discovered during primitive processing
    - MediationConfigurationException raised
    - No log written
    - Fail terminal connection followed

 © 2010 IBM Corporation

When there is a problem with something in the configuration of a mediation primitive, you see the different behaviors described on the previous slide. This is because configuration parameters are checked at different times, some being checked at the initialization of the flow and others being checked during the processing of the mediation primitive. For example, assume you have a database lookup primitive that has its fail terminal wired. Two of the configuration parameters for a database lookup are the data source JNDI name and the database table name. The data source JNDI name is checked during the setup of the flow, therefore the result of an incorrect JNDI name is a MediationRuntimeException raised with a log written and the mediation flow terminated. However, the database table name is checked during the processing of the primitive itself. Therefore, when the MediationConfigurationException is initially raised it is caught, no log is written, and the mediation flow continues by following the wire from the fail terminal of the database lookup. Only if the fail terminal is not wired will the MediationConfigurationException be logged and the flow terminated.



When the fail terminal is wired and an exception occurs within a mediation primitive, information about the error is added to the failInfo section of the context section of the service message object. The following information is added:

The **failureString** contains a text description of the error that occurred.

The **origin** contains the name of the mediation primitive in which the exception occurred.

The **invocation Path** contains a list of every mediation primitive that was encountered in the message flow, up to and including the primitive in which the error occurred. In addition, the names of the terminals through which the message passed are also listed with each primitive.

With this information, logic in the flow following the fail terminal might be able to determine what action to take in response to the failure.

Summary

- Reviewed basic concepts of mediation primitives
  - Where they fit in the overall mediation picture
  - Types of primitives
  - Terminals, wiring and the mediation flow editor
  - Exceptions and error handling

23 Introduction to mediation primitives © 2010 IBM Corporation

In this presentation, the basic concepts of mediation primitives were reviewed. This included a description of where mediation primitives fit into the overall mediation picture and what types of mediation primitives there are and how they are classified into groups. A discussion of mediation primitives terminals was provided along with a description of wiring behavior in the mediation flow editor. An explanation of exceptions and some aspects of error handling were also provided.

IBM

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBPMv7_MediationPrimitiveIntroduction.ppt

This module is also available in PDF format at: ../WBPMv7_MediationPrimitiveIntroduction.pdf

 © 2010 IBM Corporation

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information

IBM, the IBM logo, ibm.com, CICS, IMS, and WebSphere are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Java, and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS OR SOFTWARE.

© Copyright International Business Machines Corporation 2010. All rights reserved.

 © 2010 IBM Corporation